FAULT DETECTION IN THE ADVANCED ENCRYPTION STANDARD

G. Bertoni, L. Breveglieri, I. Koren and V. Piuri

Abstract. *The* AES *(Advanced Encryption Standard) is an emerging private-key cryptographic system. Performance requirements in many applications, in particular in embedded systems like smart-cards, require a* HW *implementation of* AES*, for instance as a coprocessor or as a macrocell to be added to the embedded system. A* HW *implementation of such a cryptographic system requires in turn to consider the fault detection and fault tolerance topics. This paper studies the problem of fault detection in* AES*, using a simple single fault model. The analysis or error propagation is developed by simulation, and an architectural proposal is put forward, to implement fault detection under the single fault model. Both the data-path and the control-path of the* AES *function unit are taken into consideration.*

1. INTRODUCTION

In October 2000 the National Institute of Standards and Technology has chosen the Rijndael Algorithm for the Advanced Encryption Standard. The Rijndael AES algorithm (from now on simply AES) will become the most worldwide-used secret-key algorithm in the next years. AES is intended to replace DES and Triple-DES due to their loss of security [1],[2],[4]. It enhances the features of DES, extending them in order to work with longer keys and wider blocks of data.

Implementations of AES, both in software and in hardware, have been recently proposed [5-9], but a well-defined and widespread solution does not exist yet, and consequently, this is still a current topic for research. Some coprocessor VLSI architectures for computing AES have been proposed and evaluated at the

simulation level (see above references). These architectures are particularly optimised for embedded systems.

Fault tolerance is undoubtedly a key issue when designing a coprocessor architecture for implementing the AES algorithm. The AES algorithm is in fact considerably more complex than the DES algorithm it should replace. In this paper a study of fault detection in a generic hardware implementation of the AES is presented. First an analysis of error propagation in AES is carried out, adopting for simplicity the single faulty bit model. This analysis illustrates the behavior of AES in the presence of faults. Second, a proposal for an architectural solution capable of detecting single bit faults is described. This solution extends and completes a partial solution presented in [10], which only covers the data-path of the AES algorithm, ignoring the control part of AES, which is as important as the data-path part.

The paper is organized as follows. In Section 2 a brief explanation of the Rijndael AES algorithm is reported. In Section 3 some considerations of fault tolerance in the AES algorithm are discussed. Section 4 presents some architectures for fault detection and Section 5 includes conclusions and further developments.

2. AES RIJNDAEL ALGORITHM

AES Rijndael is a symmetric block cipher algorithm, meaning that it operates on blocks of data and the same key is used both for encryption and decryption [2].

The AES Rijndael algorithm permits all the combinations of key sizes and data block sizes among 128, 196 and 256 bits. NIST however, has restricted the

length of the data blocks for the AES algorithm to 128 bits only, while the key size can be chosen among 128, 196 and 256 bits.

The AES algorithm has an iterative structure and works in rounds: a fixed set of operations, called a round, is iteratively applied to the input block of data, called state matrix, the elements of which are bytes. After iterating a predefined number of rounds the state matrix is output.

The number of rounds to be executed is determined by each key size. The use of 10 rounds requires a key of 128 bits, 12 rounds a key of 196 bits and 14 rounds a key of 256 bits.

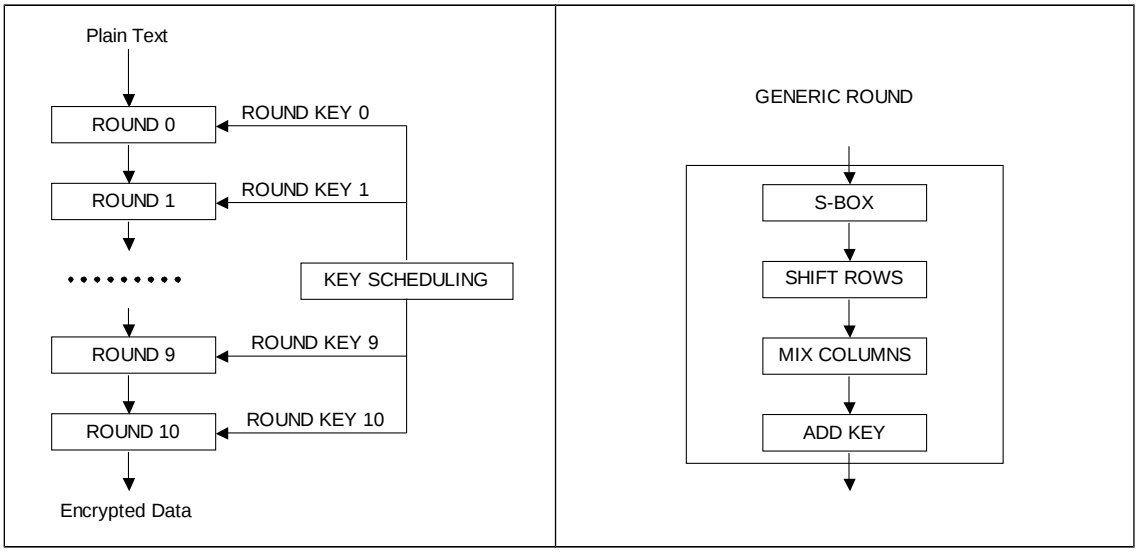

Figure 1: Logic scheme of Rijndael algorithm with 128 bits key size.

The input block of data is encrypted by applying the 10, 12 or 14 rounds. Except for the first and the last round, the other rounds are identical and consist of four transformations each. A brief overview of the algorithm, depicted in Figure 1, is given below, but the reader should refer to [2] for a complete description.

The four transformations forming each round are called SubByte, ShiftRow, MixColumn and KeyAddition. The SubByte transformation receives as input each element (one byte) of the state matrix and calculates as output its multiplicative inverse in the finite field GF(2^8), using a predefined irreducible polynomial as field generator.

The ShiftRow transformation applies a left rotation to the four rows of the state matrix: no rotation for the 1st row, a rotation of one position (one element) for the 2nd row, of two positions for the 3rd, and of three positions for the 4th. The MixColumn transformation treats each column as a 4-term polynomial, the coefficients of which are defined over the field GF(2^8), and multiplies each column by a constant polynomial, again defined over the field GF(2^8). The KeyAddition transformation adds the round key to the state matrix. The difference between the middle rounds and the first and the last ones is that the first round is composed only of a key addition, while the last one is similar to a normal round except that the MixColumn transformation is omitted.

To decrypt an enciphered block the inverse transformation should be applied, and of course the secret key must be known in advance.

The round key is derived from the secret key by means of a special algorithm, called Key Scheduling. The basic idea of Key Scheduling is that eleven different round keys of 128 bits each, are derived from a single secret key of 128 bits, for a total of 1408 bits of so-called "key material". Basically, there are two methods for calculating the round keys: complete key unrolling and the so-called “on-the-fly” approach. Figure 2 shows the C pseudocode of the key scheduling algorithm

for the complete unrolling case. The input data is the secret key, stored in the variable **key**, while the output (the key material) is stored in vector **w**.

```
KeyExpansion(byte key[4 * Nk], word w[4 * (Nr + 1)], Nk)
begin
  i=0
  while (i < Nk)
    w[i] = word[key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]]
    i = i + 1
  end while
  i = Nk
  while (i < 4 * (Nr + 1))
    word temp = w[i - 1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp)) xor Rcon[i / Nk]
    else if (Nk = 8 and i mod Nk = 4)
      temp = SubWord(temp)
    end if
    w[i] = w[i - Nk] xor temp
    i = i + 1
  end while
end
```

where

Nk is the number of secret key 32-bits words
Nr is the number of rounds
Rcon is a constant

Figure 2: Key scheduling algorithm.

The round key is composed of four 32-bit words. Inspecting the C pseudocode of the algorithm, it is possible to see that each word composing the current round key is calculated from the value of the previous 32-bit word of the current round key and from the value of the 32-bit word in the same position (1, 2, 3 or 4) of the previously computed round key. This computation is invertible, therefore, from the current round key it is possible to reconstruct the previous round key. Invertibility is necessary for an efficient “on-the-fly” key scheduling during

decryption, and will also be exploited for achieving error detection. Figure 3 shows a block diagram of the key scheduling logic.

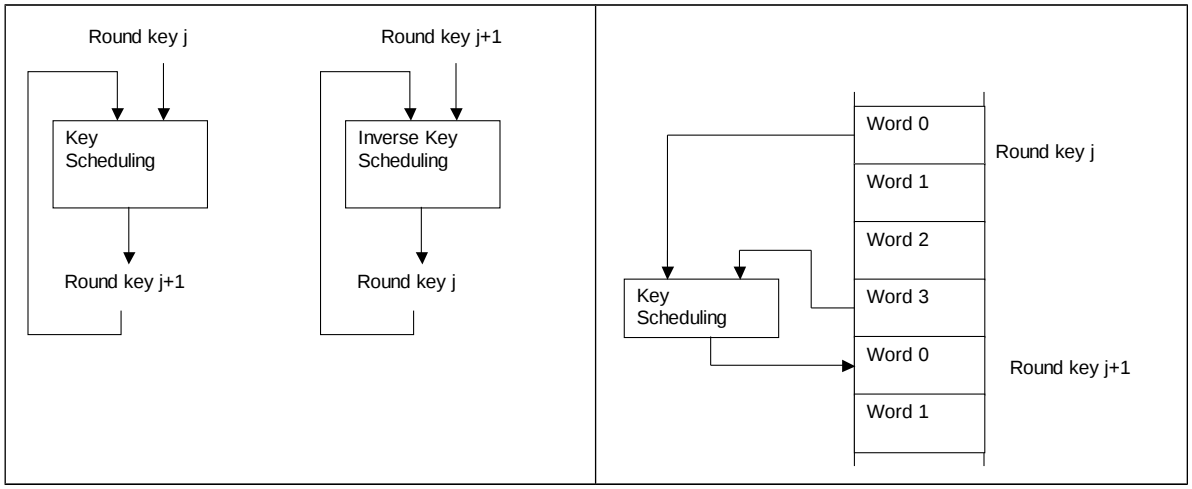

Figure 3: Logic diagram of key scheduling.

3. FAULT INSERTION

As we have seen in Section 2, the AES algorithm is basically divided into two parts: the key scheduling, which uses the secret key to obtain the round keys, and the data-path, which manipulates the plain text employing the round key to obtain the encrypted message (and vice versa for decryption). In some sense, key scheduling can be considered as the control part of the AES algorithm. In this section the error propagation behavior of the data-path is studied. The purpose of this study is to understand the effect of a fault occurring during the execution of the algorithm, on the final result. This is an important first step when developing fault detection and tolerance schemes. We adopt here, for simplicity, the single faulty bit model, i.e., only a single bit may become faulty at any given time instant. Furthermore, since the encryption and decryption processes include

a large number of steps, we restrict ourselves to single faulty bits inserted at the beginning of each round rather than during the intermediate steps within a round. Figure 4 shows the results of simulation experiments in which a faulty bit has been inserted in the AES data-path. We have used a key size of 128 bits, but we have verified that the observed behavior is similar for the other two admissible key sizes.

Two measures are important in this simulation: the effect of a fault during the computation of the encryption on the encrypted result and on the result of the decryption, where "effect" means the number of erroneous bits caused by a single faulty bit injected at some stage into the computation.

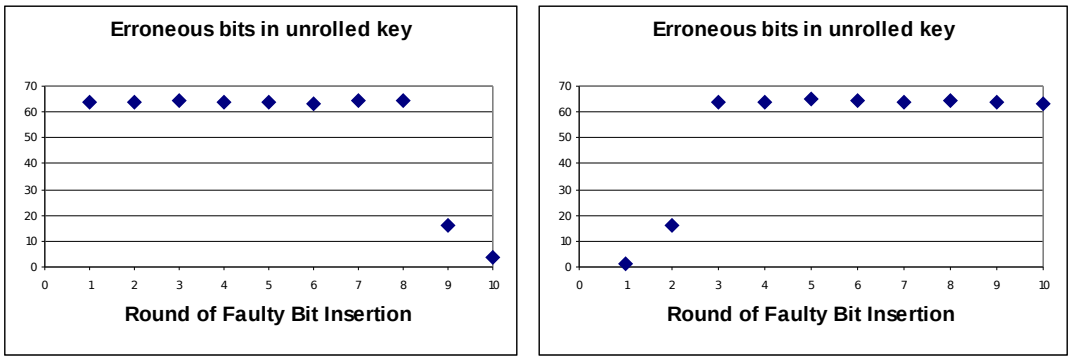

Figure 4: The mean numbers of erroneous bits in an encrypted and a decrypted data block.

From Figure 4 it is possible to see that a faulty bit inserted in the first round of encryption causes a high number of erroneous bits in the final encrypted data. Applying decryption to the corrupt data reconstructs a decrypted block containing a single faulty bit. This behavior should be expected since the AES algorithm is invertible. Still, corrupting a single bit in the input message in any round between 1 and 8 yields a corrupt encrypted message which is considerably different from

the correct one. On the average, 66 output bits were erroneous. Note that if the faulty bit is inserted in the last two rounds of encryption, it spreads over a much smaller number of bits in the final enciphered message (1 or 16 versus 66 in earlier rounds). Similarly, injecting a single faulty bit in the early rounds of decryption yields a decrypted message which is quite different from the original correct message. We have not considered inserting a faulty bit before round 0 because this would be equivalent to considering a different message.

Another part of the algorithm implementation that can be affected by faults is the Key Scheduling. We have mentioned that two methods for key scheduling are possible: a complete key unrolling and an “on-the-fly” approach. The former one can be subject to two types of errors: either a single faulty bit corrupting the stored unrolled key, or a faulty bit injected during the unrolling process which may spread to many bits. The latter method, i.e., the “on-the-fly” approach, can be subject only to the second type of error, because the unrolled key is never completely computed and stored. A faulty bit injected during the unrolling process causes a large number of erroneous bits in the next round keys.

Both situations have been simulated: a faulty bit in the stored unrolled key and a faulty bit injected during the process of unrolling the key.

The former case is equivalent to the injection of a single error in the data-path, because corrupting a single bit in the stored unrolled key has an effect identical to injecting a faulty bit in the data block at the beginning of a round, since the round key is added to the state matrix at the end of the each round.

As for the latter case, i.e., error injection and propagation during the key unrolling, we show in Figure 5 the number of erroneous bits obtained in the unrolled key, as a function of the number of the round during which the fault has been injected. In this figure, the average of the number of erroneous bits generated in the unrolled key for a secret key of 128 bits is shown, out of the total of 1408 bits composing the unrolled key material. Injecting a faulty bit in round 0 means receiving a corrupt secret key, hence this case is not considered. A faulty bit in the last unrolling round causes a single erroneous bit in the unrolled key.

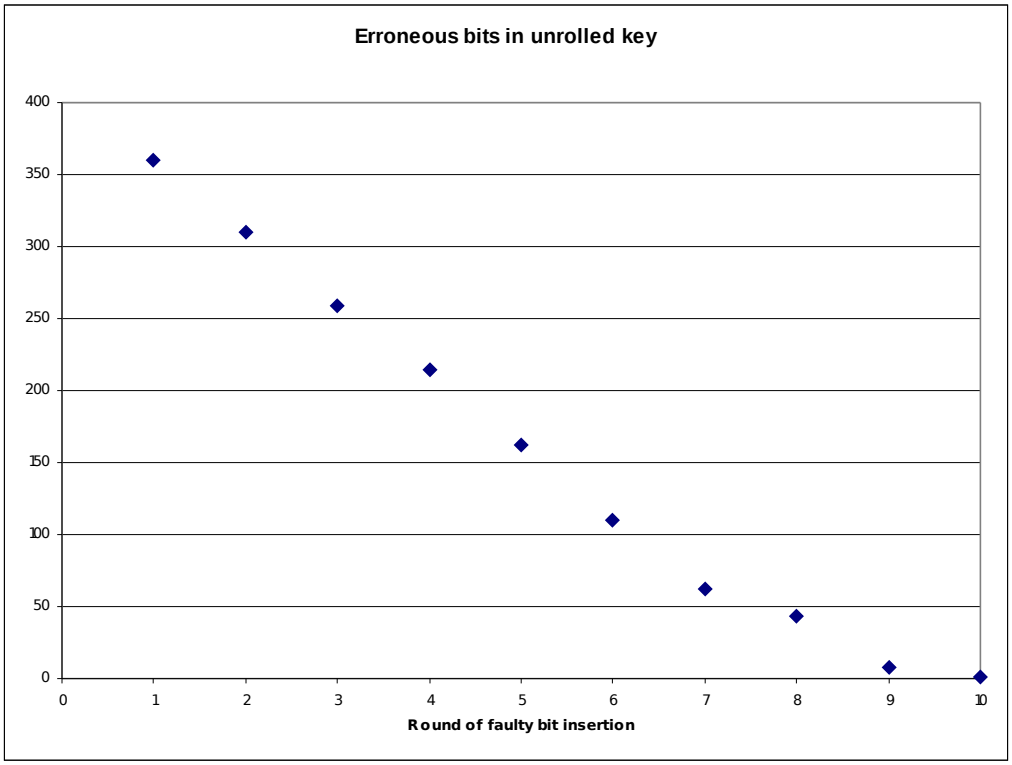

Figure 5: Propagation of a single faulty bit in the unrolling key process.

The effect of a faulty bit in the key scheduling on the data-path is shown in Figure 6. The data-path is assumed to be fault-free, while the key scheduling is affected by the injection of a single faulty bit at some unrolling round. In this case it is

possible to see that a faulty bit injected into the early rounds causes a high number of erroneous bits in the decryption process. If decryption is performed using the erroneous unrolled key to check for the presence of errors, it is not possible to detect the presence of a faulty bit in the key material. What happens is that the receiver will decrypt useless data and the sender will not be able to realize that the sent data was corrupted. This justifies the need of special attention for the management of key unrolling.

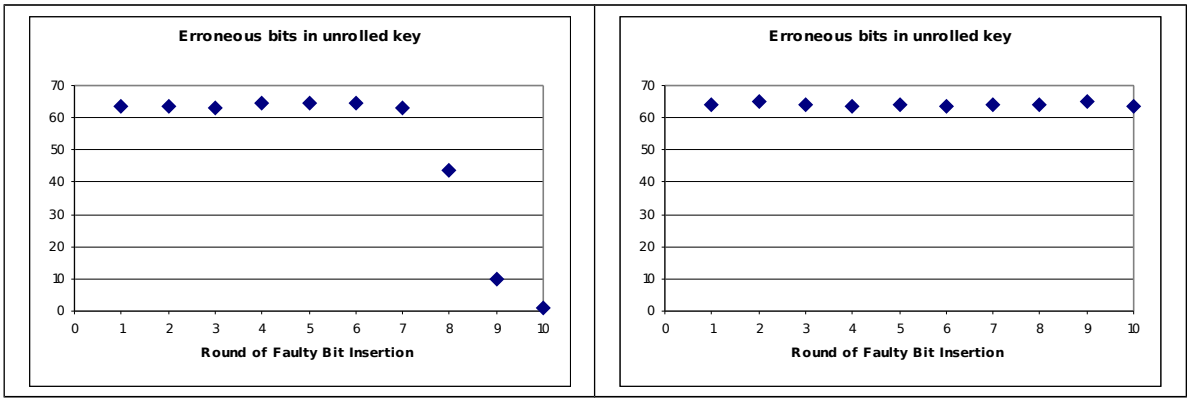

Figure 6: Effect of an erroneous unrolled key (due to a single faulty bit during the unrolling process) on the encryption and decryption processes.

4. POSSIBLE ARCHITECTURES FOR DETECTING FAULTS DURING ENCRYPTION AND DECRYPTION

As explained in the previous sections, a different behavior of the data-path and the key schedule upon the occurrence of a single faulty bit should be expected based on the above analysis of error propagation. A proposal for error detection in the data-path of AES was described in [10]. The proposed architectures were

designed to solve a different problem: one or more incorrect bits are injected by an attacker to break the cipher system. The goal of the proposed implementations was to prevent such attacks. Our objective is different; we want to detect a fault in order to prevent the transmission and use of incorrect data.

Fault detection can be applied at different levels, e.g., the encryption level, the round level or the transformation level. In the first case the whole encryption process is completed, then the enciphered data block is decrypted and the result is compared with the original plain text. If they do not match a fault in one of the processes has happened, otherwise it is reasonable to assume that the enciphered data block is correct. If a decryption core is present, the architecture overhead reduces to the cost of a comparator for two data blocks of 128 bits. The time necessary for detecting a fault is always equal to the time required to decrypt a data block, plus the time required for the comparison.

At the round level the hardware necessary to detect a fault depends on how the encryption core is implemented. For example, if the coprocessor executes one round at a time, then the necessary hardware is equal to that required by encryption plus some additional registers. If on the other hand, the entire encrypFault

Detection in the Advanced Encryption Standardtion is implemented in hardware, i.e., if for a 128-bit key size all the 10 rounds plus the initial round are individually implemented, round level fault detection becomes complex. Either one comparator plus an array of multiplexers must be implemented, or an array of 10 comparators, one for each round, should be present. If the occurrence of a fault

in a round is uniformly distributed, then the time necessary to detect a fault can be calculated as one half of the time required for encryption, plus the time required for one round of decryption and plus the time required for comparison. The time latency required for encryption depends on the implementation: if the encryption core can compute the next encryption round while the decryption core is checking the correctness of the previous round, then the time latency overhead is equal to the time necessary for one round of decryption only.

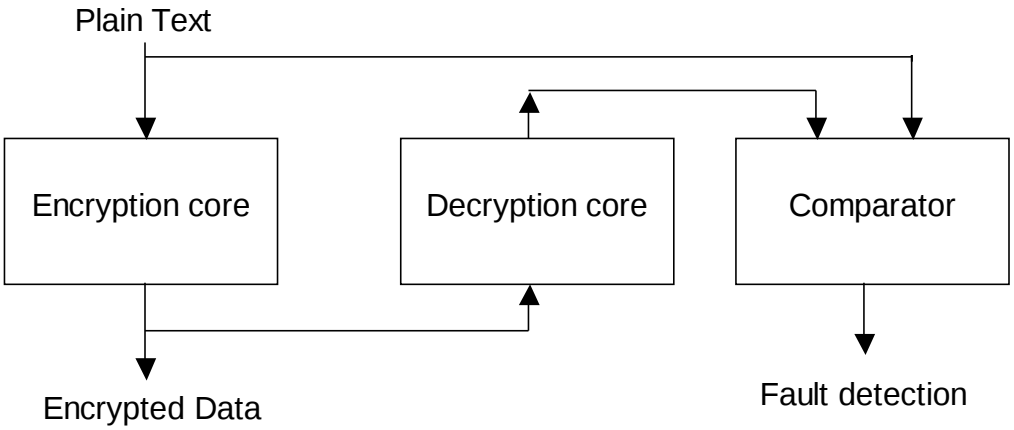

Figure 7: Fault detection at encryption level.

Both solutions are able to detect errors, but the time required to detect a faulty bit is different. A finer error detection, able to inspect the internals of each one of the four transformations forming one round (SubByte, etc), would allow a faster reaction, thus avoiding to execute useless operation on already corrupted data, but could require a large overhead in terms of components, according to the implementation. Moreover, comparison should be done at the transformation level.

As already seen in the previous section, key scheduling deserves particular care since a fault in this part of the algorithm is not detected by the techniques used for detecting faults in the data-path. We suggest to use a special key scheduling block, that could be called "inverse key scheduling", which allows to reconstruct the previous round key starting from the current one. This block is normally present in the implementation of AES adopting the “on-the-fly” key schedule method, because it is used to decrypt data blocks. In the case of a complete unrolled key, this block would not be normally present, but its implementation only causes a limited size overhead, in comparison to the complete key unrolling method. In fact, storing a completely unrolled key would require a very large register, which can be saved by adopting the on-the-fly method.

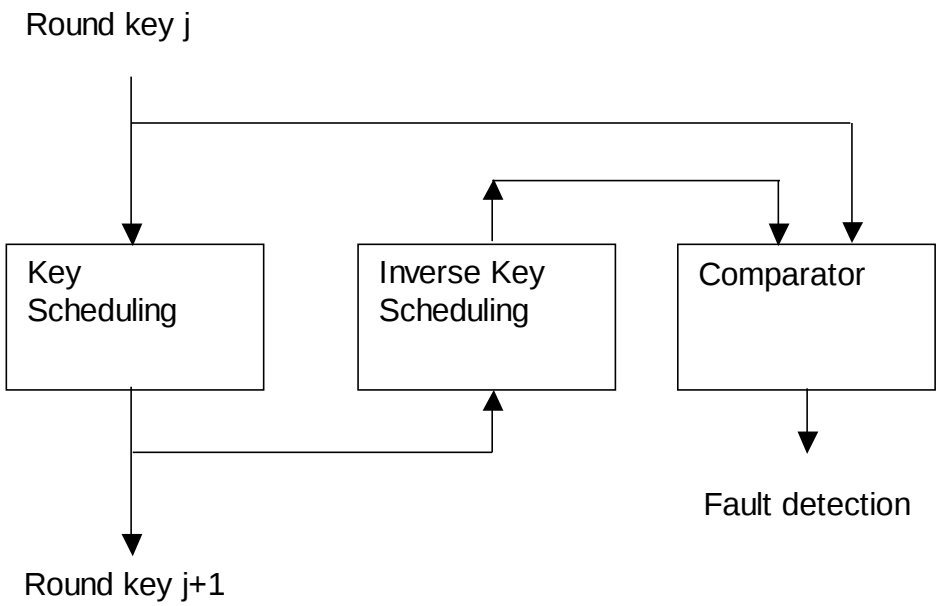

Figure 8: Logic scheme of the inverse key scheduling.

The technique is similar to the one used in the data-path: the next round key is calculated, then it is processed by the inverse key schedule block, and if the

result of the inverse key schedule matches the previous key, the computation is correct and can proceed, otherwise a fault has been detected.

5. CONCLUSION

A technique for fault detection in hardware implementations of the Advanced Encryption Standard has been studied. Simulations of the behavior of fault propagation in the encryption and decryption processes are reported and explained. The simulation proves that both parts of the algorithm, data-path and control, should be protected. Previous studies [10] have only considered the data-path, ignoring the key scheduling. A proposal for fault detection in key scheduling is presented, which may require a limited amount of circuit overhead, depending on the implementation of the system. Future research directions involve the study of fault tolerance techniques and architectures.

REFERENCES

[1] J. Daemen, V. Rijmen, “The Block Cipher Rijndael”, Smart-Card Research and Applications, LNCS 1820, J.-J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000, pp. 288-296.

[2] B. Gladman, "A Specification for Rijndael, the AES Algorithm", http://fp.gladman.plus.com/, 2001.

[3] D. Whiting, B. Schneier, S. Bellovin, "AES Key Agility Issues in High-Speed IPsec Implementations", Counterpane Internet Security, http://www.counterpane.com/aes-agility.html , 2000.

[4] M. Akkar, C. Giraud "An Implementation of DES and AES, secure against some Attacks" Proceeding of CHES 2001, Page(s): 315-325.

[5] M. McLoone, J. V. McCanny "High Performance single-Chip FPGA Rijndael Algorithm implementations" Proceeding of CHES 2001, Page(s): 68-80.

[6] V. Fischer, M. Drutarovsky "Two Methods of Rijndael Implementation in reconfigurable Hardware" Proceeding of CHES 2001, Page(s):81-96.

[7] H. Kuo and I. Verbauwhede "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm" Proceeding of CHES 2001, Page(s): 53-67.

[8] Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, "Efficient Rijndael Encryption Implementation with composite Field Arithmetic" Proceeding of CHES 2001, Page(s): 175-188.

[9] A. Dandalis, V.K. Prasanna, J.P.D. Rolim "An adaptive Cryptographic Engine for IPSec Architecutures" Field-Programmable Custom Computing Machines, 2000, IEEE Symposium on, 2000 Page(s): 132-141.

[10] R. Karri, W. Kaijie, P. Mishra,K. Yongkook, "Fault-based side-Channel Cryptanalysis tolerant Rijndael symmetric Block Cipher Architecture", Defect and Fault Tolerance in VLSI Systems, 2001 Page(s): 418 –426.